Parallel Traveling Salesperson

In this assignment, you will implement a GPU program that parallelizes a brute-force solution to the traveling salesman problem.

1 Background

The traveling salesperson problem (TSP) involves a list of cities separated by known distances. A salesperson has a customer in each city and must undertake a regular "tour," visiting the customer in each city.

To keep costs low, the salesperson wants to organize his or her trip such that he or she travels the *least total distance*, visiting each city *once* and returning to the *starting city*.

The TSP is a classic in computer science. It is an NP-hard problem, meaning, in part, that there is no known algorithm that solves it in polynomial time (i.e., in time $O(n^k)$ for any constant k). Importantly, no one has been able to prove that there *is no* polynomial-time algorithm, but none has so far been discovered. The greatest unanswered question in computer science can be simplified to asking whether (1) there *is* a polynomial-time algorithm for the TSP (or one of many similarly ambiguous problems) or (2) it can be proven that no such algorithm exists.

Lacking a polynomial-time algorithm for the TSP, we will approach the problem using brute force. Given a set of cities and the distances between each pair of cities, our algorithm will:

1. Enumerate *all* possible tours (orders of cities) for the salesperson's visits.
2. Calculate the cost of each tour as the total distance traveled.
3. Identify the cost of the cheapest tour (or tours) and (optionally) the ordered list of cities on the tour(s).

For a given set of cities, the possible tours correspond to the *permutations* of cities (not just *combinations*—the order of the cities visited matters). Because a list of n cities has $n!$ permutations, the complexity of a brute-force solution is $O(n!)$ or *exponential*. Consequently, the brute-force algorithm doesn't scale past a few dozen cities before the run-time of the algorithm will exceed the duration until the heat death of the universe.[1] We will satisfy ourselves with modest-sized TSP's and will apply the power of GPU computing to reduce the time it takes to calculate optimal tours.

2 Sample Code

Find sample code on the course web site. At your discretion, you may use any of this code to formulate your parallel implementation.

[1] Not kidding.

2.1 Sequential Implementation

The `tsp-serial` program is a serial implementation of the brute-force TSP algorithm. It accepts two command-line arguments that you may find useful:

1. `-c` sets the number of cities to include in the tour.
2. `-s` sets a seed for the random number generator that determines the distances between cities. Although the program runs fine without this option, supplying a value causes the random number generator to output the same sequence of distances, making it easier to debug your code across runs.

Try running the program with varying numbers of cities. Even on a laptop, the program runs in a few seconds or less up to 10–12 cities. Above a dozen or so, it starts to slow down considerably, suggesting the utility of a GPU solution.

Most of the work in this implementation gets done in the call to `permutations`. It takes as arguments:

1. An array `v` of integers that represent city identifiers
2. An integer `n` giving the length of `v`
3. A function `action` with the following type:[2]

Action Function Declaration

```
1 typedef void (*perm_action_t)(int *v, int n);
```

The `action` function expects an array of integers (`v`) and the array's length (`n`). The `permutations` function arranges to invoke `action` for each permutation of the `n` cities in `v`.

The actual generation of permutations is done by the recursive `perms` function, which has a signature identical to `permutations`. It's here that the `action` function is invoked after `perms` generates the "next" permutation.

When `main` calls `permutations`, it passes `eval_tsp` as the `action` function. This function evaluates the total distance of the tour given a permutation of the cities.

Where do the distances come from? The `eval_tsp` function invokes `create_tsp`, which generates distances. (It's here where the random seed from the command line is used.) Distances are stored in an `n`-by-`n` array in which element `[i][j]` stores the distance from city `i` to city `j`. It stores the same distance for the reverse direction. (We could allow a different "distance" for the reverse direction, reflecting, for example, a different "cost." For example, perhaps flights from i to j are more expensive than from j to i.) We use the `TSP_ELT` macro to access the distance array throughout the code.

Returning to `eval_tsp`, note that the return value from `create_tsp` is stored in a integer pointer declared to be `static`. Although `static` variables are visible only within function scope, they are *not* stored on the run-time stack; they persist from one invocation of the function to the next.

The `if` statement that wraps the call to `create_tsp` ensures that we only allocate the distance array once. After that, `distances` has a non-`NULL` value and retains its value throughout the execution of the program.

[2] This `typedef` declares a pointer to a function that takes and pointer to an `int` and an `int` as arguments and returns nothing (`void`).

Each time `perms` invokes `eval_tsp`, the latter function receives a new permutation of cities. It iterates over the permutation looking up the distances between cities and calculating the total distance, including the return to the first city. It compares this distance with a global `shortest_length`. If a shorter tour is found, it updates `shortest_length` and outputs the permutation and its total distance.

2.2 Ordered Permutations

Following the *PCAM* model, the key design decision we must make when parallelizing any algorithm is how to *partition* the problem. In our brute-force TSP, the basic operation is calculating the distance for a particular permutation of a list of cities (i.e., for a particular tour).

An issue for parallelization is how we will partition the permutations among the threads in the GPU. We could use a strategy similar to the sequential version with one thread (the "master") generating permutations and making them available to other threads (the "workers"), which calculate the distance. However, this approach creates bottlenecks in computation (one master) and memory (sharing permutations with workers).

k	Perms		
1	0	1	2
2	0	2	1
3	1	0	2
4	1	2	0
5	2	0	1
6	2	1	0

Table 1: Permutations for a three-city tour.

Table 1 shows the permutations of a tour of three cities. As expected, there are $3! = 6$ permutations, which are *ordered* on index k in the table. Imagine we had functions that compute:

1. The permutation at index k directly
2. The "next" permutation following a given permutation

We could then allocate k/p permutations to each of p threads. Each thread could *independently* calculate its own "starting" permutation and the permutations that follow it. There would be no need for a master or workers, nor would we pay for the bottlenecks that architecture carries.

The `kth-perm` program includes implementations of the aforementioned functions.

1. The `kth_perm` function generates the `k`-th permutation of `size` integers. It allocates an array of integers on the heap (using `malloc`) containing the permutation and returns it to the caller. (The caller should eventually `free` this array.)
2. The `next_perm` function takes an array of integers (`perm`) and its size (`size`) and generates the "next" permutation. The function does this "in place" so does not allocate additional memory.

These functions work together as follows:

Generating Permutations for a Thread

```
int my_tour = kth_perm(my_first_k_idx, num_cities);
while (/* More cities for this thread */) {
  /* Do stuff with this permutation */
  next_perm(my_tour, num_cities);
}
free(my_tour);
```

3 Code

Write a CUDA parallel program that implements a brute-force solution to the TSP. Guidelines:

1. Use the ordered permutation approach outlined in Section 2.2.
2. Implement additional command-line parameter(s) that allow you to specify the CUDA configuration (e.g., number of threads) for each run. Don't hard wire the run-time configuration, which will require recompiling for each experiment.
3. Each thread should calculate a local minimum distance for its partition of the TSP permutations.
4. After all threads complete, the host should compute and report the global minimum distance.
5. (**Extra Credit**) Update your implementation to collect and print the ordered list of cities in the lowest-cost tour(s). There more be more than one tour with the same lowest cost.

4 Experiment

Run your code for varying combinations of tour length (i.e., number of cities) and thread count. Record the time for each run (t_p).

1. Vary the number of threads from one to the total number of threads available on your GPU, doubling the thread count as you go.
2. For the number of cities, start with a number that's small enough so that t_p on one thread is near zero. Increase by one city at a time as long as the total number of permutations does not cause an overflow. The output from the `kth-perm` program can help you determine this value.
3. At your discretion, you may omit combinations of small thread counts and large numbers of cities should the run-time exceed some reasonable minimum (several minutes).

5 Report

As for previous assignments:

1. Calculate S and E for each experiment.
2. Create a table of t_p, S, and E for each combination of thread and city count.
3. Plot the same data on one or more charts.
4. Briefly report your observations on the behavior and performance of your implementation.

If you are so inclined, use LaTeX to write your report. All the cool kids are doing it.

6 Submit

Create a zip file or tar ball containing:

1. All the *source* code for your parallel implementation from Section 3. Do *not* include:
   - Any object files or executables.
   - Code you downloaded from the course web site
   - This document.
2. Your report from Section 5.

Submit the file to the course web site.